Bartoli, V., Dixon, D., & Gorochowski, T. (2018). Automated Visualization of Genetic Designs Using DNAplotlib. In J. Braman (Ed.), *Synthetic Biology: Methods and Protocols* (pp. 399-409). (Methods in Molecular Biology; Vol. 1772). Humana Press. https://doi.org/10.1007/978-1-4939-7795-6_22

Automated visualization of genetic designs using DNAplotlib

Vittorio Bartoli, Daniel O.R. Dixon and Thomas E. Gorochowski

Abstract

Visualization of complex genetic systems can help efficiently communicate important design features and clearly illustrate overall structures. To aid in the creation of such diagrams, standards like the Synthetic Biology Open Language Visual (SBOLv) have been established to ensure that specific symbols and shapes convey the same meaning for genetic parts across the field. Here, we describe several ways that the computational tool DNAplotlib can be used to automate the generation of SBOLv standard-compliant diagrams covering simple genetic designs to large libraries of genetic constructs.

Keywords: visualization; genetic design; standardization; SBOLv; synthetic biology; systems biology.

1. Introduction

Synthetic biology aims to apply engineering principles to biology, introducing the concepts of abstraction, modularization and standardization to aid in the creation of biologically-based systems with novel functionality. Several major efforts have been established to support these efforts. One of the most prominent is the Synthetic Biology Open Language (SBOL) developed to provide a standardized way to describe, store and exchange biological design information [1]. Tools that adopt SBOL can seamlessly exchange information, allowing for the creation of complex workflows that can span multiple design tools and enable many research groups to collaborate effectively [2, 3].

More recently, this standard has been complimented by SBOL Visual (SBOLv), a set of agreed upon symbols and rules to create coherent visualizations of biological designs [4]. As with other more mature engineering fields, such as electrical engineering, the ability to graphically represent elements of a system in a commonly defined way greatly improves the communication of both design principles and the overall structure of a system. Many computational genetic design tools have begun to adopt SBOLv [5–10]. However, these have tended to restrict the ability of users to customize aspects of their design such as the color and shape of symbols and the overall layout.

To address this limitation, we developed a computational library called DNAplotlib [11]. This allows for users to write visualization scripts in the Python programming language [12, 13] or use built in scripts to rapidly generate highly customizable and standard-compliant genetic diagrams from data in spreadsheets. The ability to directly access DNAplotlib through a programming language has also led to it being integrated into several other genetic design tools, such as Cello [14]. Here, we

describe several different ways that DNAplotlib can be used to generate genetic diagrams and the ways that they can be easily customized for specific requirements.

2. Materials

2.1 Dependencies

1. Several applications and supporting libraries must be available to install and use DNAplotlib. These include: Python 2.7 or later, and matplotlib 1.8 or later. We recommend using a packaged Python distribution such as Anaconda (https://www.continuum.io) or Enthought (https://www.enthought.com), which includes all the necessary dependencies by default.

2. Optionally, to allow for the reading of genetic design information from SBOL files, the pySBOL 2.0 or later library must also be present (for further details regarding installation see: https://github.com/SynBioDex/pySBOL).

2.2 Installation

1. DNAplotlib is distributed via the Python Package Index (PyPI) and uses the Pip Installs Packages (PIP) package management system to install and handle updates to the software. The pip system is included by default with Python 2.7.9 and later, and Python 3.4 and later. If a working version of a Python environment is available (see §2.1), then the latest stable release of DNAplotlib can be installed by running:

```
pip install dnaplotlib
```

2. To test that the installation has been successful, the following commands should successfully run without error:

```
python
>>> import dnaplotlib
```

3. Methods

In the following sections, we detail a range of ways that DNAplotlib can be used to generate genetic design visualizations. We focus on the use of built-in scripts to simplify the plotting of basic designs (Quick Plot) and the generation of diagrams for entire design libraries from data contained in spreadsheets (Library Plot). To learn more about the advanced features that directly call internal functionality using Python scripts, we recommend consulting our previous publication [11] and exploring the documentation and examples at the project website: www.dnaplotlib.org.

3.1 Quick Plot

1. Quick Plot is the fastest and easiest way to generate a figure with DNAplotlib. A single command can generate basic designs incorporating 7 common types of genetic part in 14 different colors (see Figure 1 for an example).

2. To use Quick Plot, the 'quick.py' script must be accessible from the command line. We recommend downloading the latest version from the 'apps' folder at www.dnaplotlib.org and either: 1. placing it in a central location and adding this to the user's PATH environment variable, or 2. placing it directly in the current working directory (see Note 1).

3. A design is specified by a single line of text that is composed of individual elements for each part to be displayed. Each part is defined by a part type (a single letter or symbol), a dot and then the color that the part should be drawn in (Figure 1). For example, a red promoter would be defined as 'p.red'. To display a part in a reverse orientation, a minus symbol is placed directly before the part type. Thus, a red promoter in a reverse direction is defined by '-p.red'. A full design consists of a sequence of these part definitions separated by spaces.

4. Once the text for a design has been produced, this is passed to the 'quick.py' script that will render an image of the construct. The script takes two arguments: 1. '-input' defines the design of the construct, and 2. '-output' provides the output filename for the visualization. It is important that the input design is encased in quotes to ensure that spaces are interpreted correctly, and the extension of the output file (e.g. pdf, png, etc.) will define the type of file that is produced (see Note 2). The command used to generate Figure 1 is shown below:

```
python quick.py -input "=.red p.green i.black r.black c.orange t.black -
t.black -c.green -r.gray -i.black -p.blue -s.lightblue =.red s.orange
p.orange r.gray c.blue t.black" -output QuickPlot.pdf
```

3.2 Library Plot

1. Library Plot enables the plotting of many genetic designs from data contained within spreadsheets (see Figures 2 and 3 for some examples). Information about parts, regulation, and designs is stored in separate spreadsheets and Library Plot uses this information to generate a combined plot of them all. Because separate spreadsheets are used for part and design information, the parts spreadsheet can be easily reused and shared across many different plots. For example, the same parts spreadsheet can be used by everyone in a lab to ensure the

formatting of genetic elements is consistent. Some examples of the required spreadsheets and their formats are available from the project website: www.dnaplotlib.org.

2. To use Library Plot, the 'library_plot.py' script must be accessible from the command line. We recommend downloading the latest version from the 'apps' folder at www.dnaplotlib.org and either: 1. placing it in a central location and adding this to the user's PATH environment variable, or 2. placing it directly in the current working directory (see Note 3).

3. Next, a spreadsheet must be created that contains all the parts that are featured in the plot. We recommend calling this file 'parts.csv' and it must be saved in a comma-separated values (CSV) format. This spreadsheet should have columns for each of the options listed in Table 1 with a header row containing each option's name. Each row under this header defines a part that can be later used. It is essential that the 'part_name' option is filled in as this will be referred to in the design and regulation spreadsheets. The 'type' option must also be specified. Table 1 provides details of every option and the formats that are accepted.

3. A similar spreadsheet should be created that contains all the designs to be plotted. We recommend calling this file 'designs.csv' and it must be saved in a comma-separated values (CSV) format. The first row will be ignored, but we recommend using the headings 'design_name' for the first column and 'parts' for the second. Each row under this header then defines a design that will be processed. For each design, a 'design_name' must be given in the first column (see Note 4), and then the proceeding cells define the order of the parts making it up. The 'part_name' should be used to define the element at each position and to plot parts in reverse orientation, the letter 'r' should be placed directly before the name of the part. For example, if the part is called 'RBS1', then 'rRBS1' would be entered to plot the part in a reverse orientation.

4. (Optional step) If regulation is present in any of the designs, a third regulatory spreadsheet should be created. We recommend calling this file 'regulation.csv' and it must be saved in a comma-separated values (CSV) format. This spreadsheet should have columns for each of the options listed in Table 2, with a header row containing each option's name. Each row under this header defines a regulatory arc. It is essential that the 'from_partname' and 'to_partname' options are present and refer to parts present in the parts spreadsheet (see Note 5). These

options define the source and target of the regulatory arc, respectively. The 'type' option must also be specified as either Activation, Repression or Connection (see Note 6).

5. The final spreadsheet to be created contains general parameters that influence the overall plotting of the designs. We recommend calling this file 'parameters.csv' and it must be saved in a comma-separated values (CSV) format. This spreadsheet should have columns for each of the options listed in Table 3, with a header row containing each option's name. Each row under this header defines a parameter setting. For every parameter, a value must be set.

6. Once these spreadsheets have been produced, they are passed to the 'library_plot.py' script that will render an image of all the constructs. The script takes five arguments: 1. '-params' provides the filename of the parameters spreadsheet, 2. '-parts' provides the filename of the parts spreadsheet, 3. '-designs' provides the filename of the designs spreadsheet, 4. '-regulation' provides the filename of the regulation spreadsheet, and 5. '-output' provides the output filename for the visualization. The extension of the output file (e.g. pdf, png, etc.) will define the type of file that is produced (see Note 3). The command used to generate Figure 3 is shown below (we assume that recommended names are used for each spreadsheet):

```
python library_plot.py -params parameters.csv -parts parts.csv -
regulation regulation.csv -designs designs.csv -output LibraryPlot.pdf
```

4. Notes

1. To test that quick.py is available, type "python quick.py" at the command line. This should return details of how to use the command and not throw an error. If an error is shown then check that the quick.py script is in the current directory or is present at a directory listed in the PATH environment variable.

2. The format of the output file is determined by the file extension. Standard available file formats and their file extensions include: PGF code for LaTeX (pgf), Scalable Vector Graphics (svgz), Tagged Image File Format (tif or tiff), Joint Photographic Experts Group (jpg or jpeg), Raw RGBA bitmap (raw), Portable Network Graphics (png), Postscript (ps), Scalable Vector Graphics (svg), Encapsulated Postscript (eps), Raw RGBA bitmap (rgba), and Portable Document Format (pdf). Note that the actual image formats supported may vary due to differences in Python distributions.

3. To test that library_plot.py is available, type “python library_plot.py” at the command line. This should return details of how to use the command and not throw an error. If an error is shown then check that the library_plot.py script is in the current directory or is present at a directory listed in the PATH environment variable.

4. Designs are plotted in alphabetical order of the ‘design_name’. We recommend using a numbering format (e.g. 001, 002) as a prefix so that you can easily control the order in which parts are printed. These will not show up in the plot unless you set the ‘show_title’ option to ‘Y’ in the parameters spreadsheet (Table 3).

5. Note that regulatory arcs can only go between parts on the same design and cannot go between parts on different designs on the same library plot.

6. If regulatory arcs disappear off the top of the plot, increase the value for ‘axis_y’ option in the parameters spreadsheet (Table 3).

Acknowledgements

T.E.G. was supported by BrisSynBio, a BBSRC/EPSRC Synthetic Biology Research Centre (grant BB/L01386X/1). V.B. and D.O.R.D acknowledge funding from the EPSRC & BBSRC Centre for Doctoral Training in Synthetic Biology (grant EP/L016494/1). We also thank Mario di Bernardo and Nigel Savery for comments.

References

1. Galdzicki M, Clancy KP, Oberortner E, Pocock M, Quinn JY, Rodriguez CA, Roehner N, Wilson ML, Adam L, Anderson JC, Bartley BA, Beal J, Chandran D, Chen J, Densmore D, Endy D, Grunberg R, Hallinan J, Hillson NJ, Johnson JD, Kuchinsky A, Lux M, Misirli G, Peccoud J, Plahar HA, Sirin E, Stan GB, Villalobos A, Wipat A, Gennari JH, Myers CJ, Sauro HM (2014) The Synthetic Biology Open Language (SBOL) provides a community standard for communicating designs in synthetic biology. *Nature Biotechnology* 32: 545-550.

2. Myers CJ, Beal J, Gorochowski TE, Kuwahara H, Madsen C, McLaughlin JA, Misirli G, Nguyen T, Oberortner E, Samineni M, Wipat A, Zhang M, Zundel Z. (2017) A Standard-Enabled Workflow for Synthetic Biology. *Biochemical Society Transactions*.

3. Roehner N, Beal J, Clancy K, Bartley B, Misirli G, Grünberg R, Oberortner E, Pocock M, Bissell M, Madsen C, Nguyen T, Zhang M, Zhang Z, Zundel Z, Densmore D, Gennari JH, Wipat A, Sauro HM, Myers CJ (2016) Sharing Structure and Function in Biological Design with SBOL 2.0. ACS Synthetic Biology (6): 498-506. (DOI: 10.1021/acssynbio.5b00215)

4. Quinn JY, Cox RS III, Adler A, Beal J, Bhatia S, Cai Y, Chen J, Clancy K, Galdzicki M, Hillson NJ, Le Novère N, Maheshwari AJ, McLaughlin JA, Myers CJ, Umesh P, Pocock M, Rodriguez C, Soldatova L, Stan G-BV, Swainston N, Wipat A, Sauro HM (2015) SBOL Visual: A Graphical Language for Genetic Designs. *PLoS Biology* 13(12): e1002310. (DOI: 10.1371/journal.pbio.1002310)

5. Lu G, Moriyama EN (2004) Vector NTI, a balanced all-in-one sequence analysis suite. *Briefings in Bioinformatics* 5: 378-388.

6. Chandran D, Bergmann FT, Sauro HM (2009) TinkerCell: modular CAD tool for synthetic biology. *Journal of Biological Engineering* 3: 19.

7. Czar MJ, Cai Y, Peccoud J (2009) Writing DNA with GenoCAD. *Nucleic Acids Research* 37: W40-47.

8. Chen J, Densmore D, Ham TS, Keasling JD, Hillson NJ (2012) DeviceEditor visual biological CAD canvas. *Journal of Biological Engineering* 6: 1-12.

9. Bhatia S, Densmore D (2013) Pigeon: a design visualizer for synthetic biology, *ACS Synthetic Biology* 2: 348-350.

10. McLaughlin JA, Pocock M, Misirli G, Madsen C, Wipat A (2016) VisBOL: Web-Based Tools for Synthetic Biology Design Visualization, *ACS Synthetic Biology* 5: 874-876.

11. Der BS, Glassey E, Bartley BA, Enghuus C, Goodman DB, Gordon DB, Voigt CA, Gorochowski TE (2016) DNAplotlib: programmable visualization of genetic designs and associated data. *ACS Synthetic Biolo*gy. (DOI: 10.1021/acssynbio.6b00252)

12. Sanner MF (1999) Python: a programming language for software integration and development. *Journal of Molecular Graphics and Modelling* 17: 57-61.

13. Hunter JD (2007) Matplotlib: A 2D Graphics Environment. *Computing in Science & Engineering* 9: 90-95.

14. Nielsen AAK, Der BS, Shin J, Vaidyanathan P, Paralanov V, Strychalski EA, Ross D, Densmore D, Voigt CA (2016) Genetic circuit design automation. *Science* 352: aac7341.

Figures Captions

Figure 1: **Quick Plot options.** (**A**) Descriptions of all allowed part types and colors. The minus sign before a part type causes the part to be drawn in a reverse orientation. An example of the command to generate the diagram is shown. (**B**) Visualization generated by the design description in panel A. SBOL visual symbols are used; coding sequences are denoted by the large arrows and their expression produces a protein product that repress their cognate promoter (shown in the same color). (**C**) Three examples of simple constructs where the color and orientation of the coding region is varied.

Figure 2: **Overview of parts available when using Library Plot.** (**A**) All SBOLv parts in a forward orientation. The type of each part is labelled. (**B**) All parts in a reverse orientation. (**C**) Examples of some customization options available to alter the default shape and color of symbols. Figure adapted from Der *et al.* [11]. Accompanying spreadsheets can be found in the gallery at www.dnaplotlib.org.

Figure 3: **States of an XNOR genetic circuit visualized using Library Plot.** Colored genes correspond to repressor proteins and their cognate promoters are shown in the same color. For each state only active regulatory links are displayed. Active promoters are shown in black or are strongly colored. Genes that are expressed are filled in color with their name shown.

Tables

Table 1: Library Plot parts spreadsheet options

Option	Description	Format / Values	Default
part_name[a]	Name of part	Alphanumeric	n/a
type[a]	Type of part	Promoter | RBS | CDS | Terminator | Ribozyme | Scar | Spacer | Ribonuclease | ProteinStability | Protease | Operator | Origin | Insulator | 5Overhang | 3Overhang | RestrictionSite | BluntRestrictionSite | PrimerBindingSite | 5StickyRestrictionSite | 3StickyRestrictionSite | UserDefined | Signature | Repressor | EmptySpace[b]	n/a
x_extent	Horizontal length of part	Decimal	– [c]
y_extent	Vertical height of part	Decimal	– [c]
start_pad	Empty space at start of part	Decimal	– [c]
end_pad	Empty space at end of part	Decimal	– [c]
color	Color of part	(Red, Green, Blue)[d]	– [c]
hatch[e]	Hatch pattern type	/ | // | /// | //// | /////[f]	(none)
arrowhead_height[g]	Height of arrow head	Decimal	– [c]
arrowhead_length[g]	Length of arrow head	Decimal	– [c]
linestyle[g]	Line style	- | -- | --. | :[f]	-
linewidth	Line width	Decimal	1
fill_color[i]	Color of inside of part	(Red, Green, Blue)[d]	(1, 1, 1)
edge_color[j]	Color of part's edge	(Red, Green, Blue)[d]	(1, 1, 1)
site_space[k]	Empty space between restriction site cuts	Decimal	1.5
end_space[l]	Space either side of sticky restriction site cuts	Decimal	1
label	Label text	Alphanumeric	(none)
label_style	Label text style	normal | *italic* | **bold**	normal
label_size	Font size of label	Decimal	7
label_y_offset	Vertical label position	Decimal	0
label_x_offset	Horizontal label position	Decimal	0
label_color	Color of label text	(Red, Green, Blue)	(0, 0, 0)

a. Required option.

b. See Figure 2 for examples of each part type.

c. Dependent on part type.
d. Red, green and blue components are given in the range 0 to 1.
e. Only valid for coding region (CDS) part types.
f. Line and hatch styles follow the matplotlib format (see www.matplotlib.org for details).
g. Only valid for coding region (CDS) and Promoter part types.
h. Only valid for Ribozyme, Scar, Spacer, Ribonuclease, ProteinStability, 5Overhang, 3Overhang, RestrictionSite, BluntRestrictionSite, 5StickyRestrictionSite, 3StickyRestrictionSite and Signature part types.
i. Only valid for UserDefined and Signature part types.
j. Only valid for coding region (CDS) and RBS part types.
k. Only valid for BluntRestrictionSite part types.
l. Only valid for 5StickyRestrictionSite and 3StickyRestrictionSite part types.

Table 2: Library Plot regulation spreadsheet options

Option	Description	Format / Values	Default
from_partname[a]	Part at start of regulation arc	part_name in 'parts' spreadsheet	n/a
to_partname[a]	Part at end of regulation arc	part_name in 'parts' spreadsheet	n/a
type[a]	Type of regulation arc	Activation | Repression | Connection	n/a
arrowhead_length	Length of arc head	Decimal	4
linestyle	Style of arc line	- | -- | -. | :[b]	-
linewidth	Line width of arc	Decimal	1.0
color	Color of arc	(Red, Green, Blue)[c]	(0, 0, 0)
arc_height	Height of arc above backbone	Decimal	20
arc_height_const	Sets position of arc above backbone minus spacing	Decimal	15
arc_height_spacing	Vertical spacing between arcs	Decimal	5
arc_height_start	Vertical start position of arc	Decimal	10
arc_height_end	Vertical end position of arc	Decimal	15

a. Required option.
b. Line styles follow the matplotlib format (see www.matplotlib.org for details).
c. Red, green and blue components are given in the range 0 to 1.

Table 3: Library Plot parameters spreadsheet options

Option	Description	Format / Values	Default
linewidth	Default line width for all parts	Decimal	1
show_title	Display titles on each design	Y | N	N
axis_y	Vertical extent of each design	Decimal	35

Figure 1

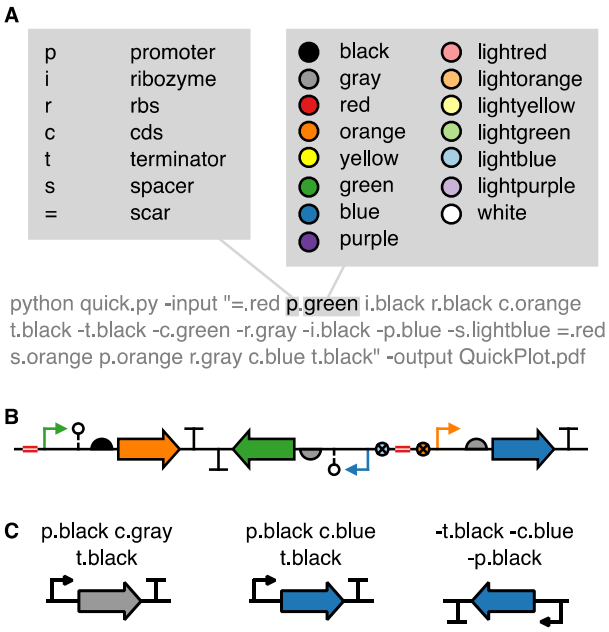

A
p promoter
i ribozyme
r rbs
c cds
t terminator
s spacer
= scar
black
gray
red
orange
yellow
green
blue
purple
lightred
lightorange
lightyellow
lightgreen
lightblue
lightpurple
white
python quick.py -input "=.red p.green i.black r.black c.orange t.black -t.black -c.green -r.gray -i.black -p.blue -s.lightblue =.red s.orange p.orange r.gray c.blue t.black" -output QuickPlot.pdf
B
C
p.black c.gray t.black
p.black c.blue t.black
-t.black -c.blue -p.black

Figure 2

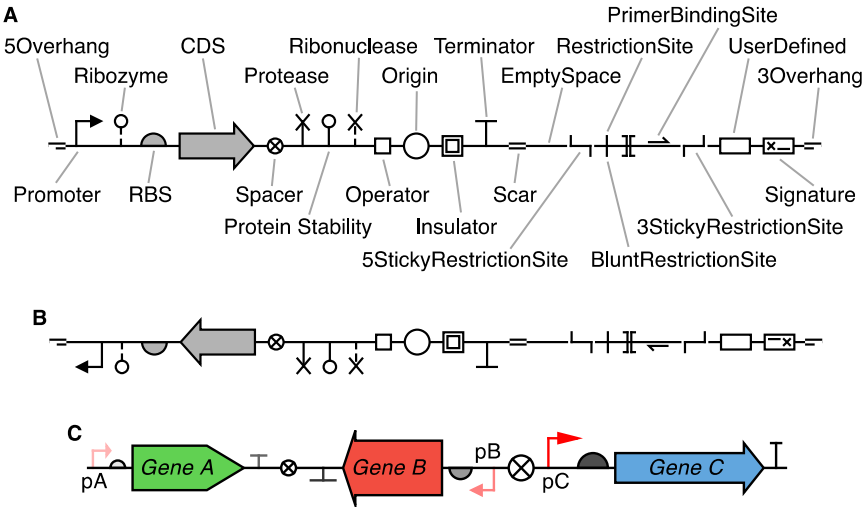
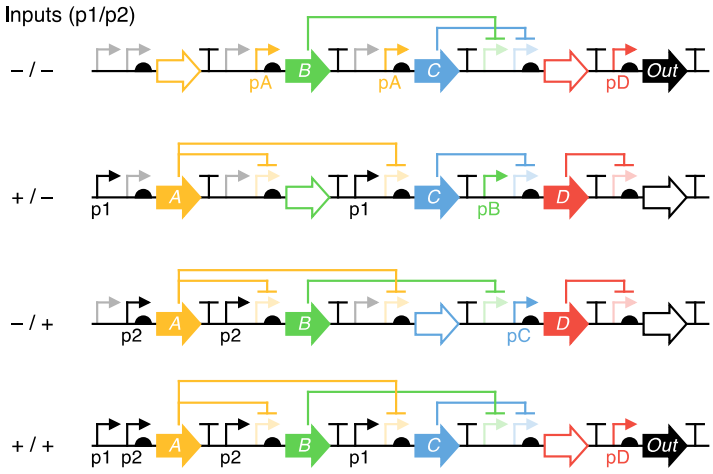

A
5Overhang
Ribozyme
CDS
Protease
Ribonuclease
Origin
Terminator
EmptySpace
RestrictionSite
PrimerBindingSite
UserDefined
3Overhang
Promoter
RBS
Spacer
Protein Stability
Operator
Insulator
Scar
5StickyRestrictionSite
BluntRestrictionSite
3StickyRestrictionSite
Signature
B
C
pA
Gene A
Gene B
pB
pC
Gene C

Figure 3


Inputs (p1/p2)
– / –
pA
B
pA
C
pD
Out
+ / –
p1
A
p1
C
pB
D
– / +
p2
A
p2
B
pC
D
+ / +
p1 p2
A
p2
B
p1
C
pD
Out